<u>Homework 1</u>

1. Problem Formulation

a) A state of the problem is defined by the 5-tuple <ML, CL, MR, CR, B>, where ML is the number of missionaries on the left bank, CL is the number of cannibals on the left bank, MR is the number of missionaries on the right bank, CR is the number of cannibals on the right bank and B is either 0 or 1, depending on whether the boat is on the left or right bank, respectively. The number of missionaries or cannibals is a number between 0 and 3.

The initial state is where all the missionaries and cannibals are on the left bank and the boat is also on the left bank: <3, 3, 0, 0, 0>.

The goal state is where all the missionaries and cannibals are on the right side and the boar is also on the right side: <0, 0, 3, 3, 1>.

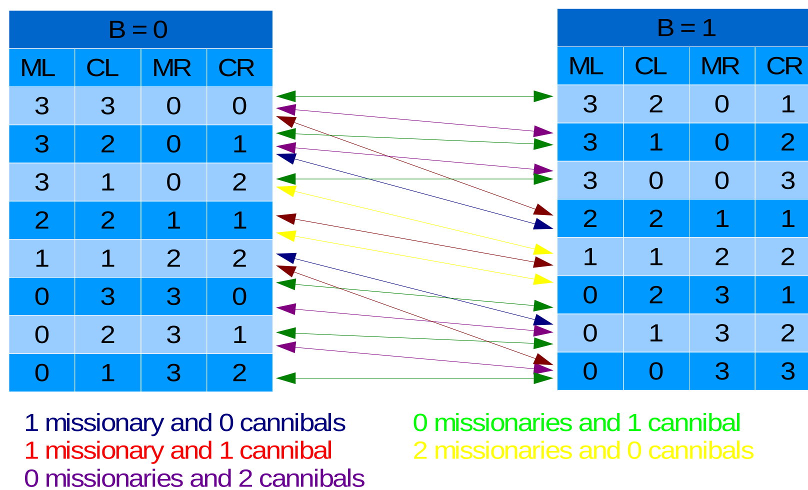
The successor function generates the legal states that result from choosing 1 or 2 missionaries or cannibals from the bank the boat is currently on and putting them in the opposite side. Let m ($0 \leq m \leq 1$) be the number of missionaries to cross the river and c ($0 \leq c \leq 1$) the number of cannibals to cross the river. The successor funcion considers every possible assignment to m and c, such that $c + m = 1$ or $c + m = 2$, and generates states according to the following equations:

if B == L then B = R; ML = ML – m; MR = MR + m; CL = CL – c; CR = CR + c;
else B = L; ML = ML + m; MR = MR - m; CL = CL + c; CR = CR – c ; fi

A state is legal iff (ML ≥ CL or ML == 0) and (MR ≥ CR or MR == 0).

The cost function is the number of times the boat is moved from one bank to the other.

b) The following figure shows a diagram of the complete state space. Arrows of different colors



represent different numbers of missionaries or cannibals that cross the river on the boat. Note that there are 4 legal states (<3, 0, 0, 3, 0>, <0, 0, 3, 3, 0>, <3, 3, 0, 0, 1> and <0, 3, 3, 0, 1>) that are not shown

because they cannot be reached from the initial state.

2. Node Generation

a) The following list assumes that nodes are expanded by considering the number of missionaries or cannibals in the following order: 1 missionary and 0 cannibals, 2 missionaries and 0 cannibal, 0 missionaries and 1 cannibal, 1 missionary and 1 cannibals, and 0 missionaries and 2 cannibals.

Cutoff	Actions (<m, c>)	State
0	initial state	<3, 3, 0, 0, 0>
1	initial state	<3, 3, 0, 0, 0>
1	<0, 1>	<3, 2, 0, 1, 1>
1	<1, 1>	<2, 2, 1, 1, 1>
1	<2, 0>	<3, 1, 0, 2, 1>
2	initial state	<3, 3, 0, 0, 0>
2	<0, 1>	<3, 2, 0, 1, 1>
2	<0, 1>, <0, 1>	<3, 3, 0, 0, 0>
2	<1, 1>	<2, 2, 1, 1, 1>
2	<1, 1>, <1, 0>	<3, 2, 0, 1, 0>

b) The first heuristic is defined as follows: the cost of moving the people to the other side of the river without the restriction that missionaries cannot be outnumbered by cannibals in either side of the river. This means that one person could take the boat back and forth between both banks until there is nobody left in the left bank. Let n be the number of missionaries and cannibals on the left bank (n = ML + CL), then the first heuristic is:

$$h_1(n) = \begin{cases} 1, & \text{if } B=0 \wedge n=1 \\ 2n-3, & \text{if } B=0 \wedge n>1 \\ 2n, & \text{if } B=1 \end{cases}$$

This heuristic is consistent, because when moving the boat from left to right, the value of the heuristic function is decremented by at most 1, which is smaller than or equal to the step cost of crossing the river, and when moving the boat from right to left the value of the heuristic function is also only decremented by at most 1. The heuristic is admissible, because it is consistent.

The second heuristic is defined as follows: the cost of moving the people to the other side of the river without the restrictions that missionaries cannot be outnumbered by cannibals and that at least one person must be on the boat when it crosses the river. This means that the boat could go back to the other bank by itself. The restriction on the maximum number of people on the boat is maitained. The second heuritic is then:

$$h_1(n) = \begin{cases} 2 \cdot \left\lfloor \frac{n+1}{2} \right\rfloor - 1, & if\, B = 0 \\ 2 \cdot \left\lfloor \frac{n+1}{2} \right\rfloor, & if\, B = 1 \end{cases}$$

This heuristic is also consistent (and hence admissible) because of the same reason that the first heuristic is consistent.

The first heuristic dominates the second one, because h_1 is greater than or equal to h_2 for all *n* and both heuristics are admissible.

c) The following list assumes that when two nodes with the same f-value are at the top of the fringe, the one that was added first is visited first. Nodes are added according to the same order used in a). Nodes are listed according to the graph-search algorithm. Nodes are visited before checking whether they are in the closed list.

State	f-value	Nodes generated and f-value of each node
<3, 3, 0, 0, 0>	9	(<3, 2, 0, 1, 1>, 11), (<2, 2, 1, 1, 1>, 9), (<3, 1, 0, 2, 1>, 9)
<2, 2, 1, 1, 1>	9	(<3, 2, 0, 1, 0>, 9), (<3, 3, 0, 0, 0>, 11)
<3, 1, 0, 2, 1>	9	(<3, 2, 0, 1, 0>, 9), (<3, 3, 0, 0, 0>, 11)
<3, 2, 0, 1, 0>	9	(<2, 2, 1, 1, 1>,11), (<3, 1, 0, 2, 1>, 11), (<3, 0, 0, 3, 1>, 9)
<3, 2, 0, 1, 0>	9	Repeated
<3, 0, 0, 1, 1>	9	(<3, 1, 0, 2, 0>, 9), (<3, 2, 0, 1, 0>, 11)
<3, 1, 0, 2, 0>	9	(<1, 1, 2, 2, 1>, 9), (<3, 0, 0, 3, 1>, 11)
<1, 1, 2, 2, 1>	9	(<3, 1, 0, 2, 0>, 11), (<2, 2, 1, 1, 1>,11)
<3, 0, 0, 1, 1>	11	Repeated
<2, 2, 1, 1, 0>	11	(<0, 2, 3, 1, 1>, 11), (<1, 1, 2, 2, 1>, 11

3. Implementation

a) Both algorithms were implemented succesfully and they work on any number of missionaries or cannibals. The parameters to the program are the following:

```
-m numMissionaries -c numCannibals -b boatCapacity -a algorithm(ids|
astar) -h heuristic(none|h1|h2)
```

IDS Algorithm

The iterative deepening search was implemented as treesearch using a recursive algorithm. The implementation is based on the code presented in the book (figures 3.9, 3.13 and 3.14).

The following code snippet shows the successor function, which considers every posible assignment to variables m (missionaries) and c (cannibals) and determines if it is a valid transition (boat has at least one person, capacity is not being exceeded and missionaries are not being outnumbered by cannibals) and if it leads to a valid state (no negative number of people on either bank and missionaries are not being outnumbered by cannibals):

```
public List<Pair<Action, State>> successorFunction(Node node){
	List<Pair<Action, State>> ret = new ArrayList<Pair<Action, State>>();
	for (int c= 0; c <= boatCapacity; c++){
		for (int m = 0; m <= boatCapacity; m++){
			if ((m >= c || m == 0) && m + c >= 1 && m + c <= boatCapacity){
				Action action = new Action(m, c);
				State oldState = node.getState();
				State newState = new State(oldState, action);
				if (newState.isValid()){
					ret.add(new Pair<Action, State>(action, newState));
				}
			}
		}
	}
	return ret;
}

public boolean isValid() {
	//First check banks have 0 more missionaries/cannibals
	if (ML >= 0 && MR >= 0 && CL >= 0 && CR >= 0){
		//Then check cannibals do not outnumber missionaries
		if ((ML >= CL || ML == 0) && (MR >= CR || MR == 0)){
			return true;
		} else {
			return false;
		}
	} else {
		return false;
	}
}
```

The goal test is presented in the following code. Since there is only one goal state, the goal test consists only of comparing the current state to this goal state.

```
public boolean isGoalState(State state){
	return state.equals(goalState);
}

public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	State other = (State) obj;
	if (B != other.B)
		return false;
	if (CL != other.CL)
```

```
            return false;
        if (CR != other.CR)
            return false;
        if (ML != other.ML)
            return false;
        if (MR != other.MR)
            return false;
        return true;
}
```

The following text presents the first and last few lines of the output of the program. It shows the nodes being generated and the nodes being expanded, as well as the statistics requested. The nodes expanded are the same obtained on part 2a). Note that the the maximum length of the fringe is not calculated because I implemented the recursive version of the algorithm, which does not keep a fringe.

```
Expanding node: [state: (3, 3, 0, 0, 0), action: null, cost: 0, depth: 0]
Generating node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Expanding node: [state: (3, 3, 0, 0, 0), action: null, cost: 0, depth: 0]
Generating node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Expanding node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Generating node: [state: (3, 3, 0, 0, 0), action: (0, 1), cost: 2, depth: 2]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Expanding node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Generating node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Generating node: [state: (3, 3, 0, 0, 0), action: (1, 1), cost: 2, depth: 2]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Expanding node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Generating node: [state: (3, 2, 0, 1, 0), action: (0, 1), cost: 2, depth: 2]
Generating node: [state: (3, 3, 0, 0, 0), action: (0, 2), cost: 2, depth: 2]
Expanding node: [state: (3, 3, 0, 0, 0), action: null, cost: 0, depth: 0]
Generating node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Expanding node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Generating node: [state: (3, 3, 0, 0, 0), action: (0, 1), cost: 2, depth: 2]
Expanding node: [state: (3, 3, 0, 0, 0), action: (0, 1), cost: 2, depth: 2]
Generating node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 3, depth: 3]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 3, depth: 3]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 3, depth: 3]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Expanding node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Generating node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Expanding node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 0), cost: 3, depth: 3]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 1), cost: 3, depth: 3]
Generating node: [state: (3, 0, 0, 3, 1), action: (0, 2), cost: 3, depth: 3]
Generating node: [state: (3, 3, 0, 0, 0), action: (1, 1), cost: 2, depth: 2]
.
.
.
Expanding node: [state: (2, 2, 1, 1, 0), action: (1, 1), cost: 10, depth: 10]
Expanding node: [state: (0, 3, 3, 0, 0), action: (0, 1), cost: 8, depth: 8]
Expanding node: [state: (0, 2, 3, 1, 1), action: (0, 1), cost: 9, depth: 9]
Expanding node: [state: (2, 2, 1, 1, 0), action: (2, 0), cost: 10, depth: 10]
Expanding node: [state: (0, 3, 3, 0, 0), action: (0, 1), cost: 10, depth: 10]
Expanding node: [state: (0, 1, 3, 2, 1), action: (0, 2), cost: 9, depth: 9]
Expanding node: [state: (1, 1, 2, 2, 0), action: (1, 0), cost: 10, depth: 10]
```

```
Move 1 missionaries and 1 cannibals to the right side
Move 1 missionaries and 0 cannibals to the left side
Move 0 missionaries and 2 cannibals to the right side
Move 0 missionaries and 1 cannibals to the left side
Move 2 missionaries and 0 cannibals to the right side
Move 1 missionaries and 1 cannibals to the left side
Move 2 missionaries and 0 cannibals to the right side
Move 0 missionaries and 1 cannibals to the left side
Move 0 missionaries and 2 cannibals to the right side
Move 1 missionaries and 0 cannibals to the left side
Move 1 missionaries and 1 cannibals to the right side
Total number of moves: 11
Depth of the solution: 11
Maximum length of the fringe: 0
Number of nodes generated: 20751
Number of nodes expanded: 9010
```

A* Algorithm

The A* search was implemented as a graph search with a priority queue ordered by f-value as the fring and a hash set for the closed list. The implementation is based on the code presented in the book (figure 3.15?). Based on point 2), it was determined that only 2 o 3 different actions (one of which leads back to the parent's state) can be performed in each state. In addition, only a few different states can be reached from the initial state. Since there could be many repeated states in the search tree, I decided to implement graphsearch. I did not, however, implement searchtree for the iterative deepening search, because it was not required that it found an optimal solution and treesearch seemed easier.

The following code shows the implementation of the heuristic functions, which follows precisely the definition presented above.

```java
public int evaluate(State state) {
	int B = state.getB();
	int n = state.getML() + state.getCL();
	if (B == 0 && n == 1){
		return 1;
	} else if (B == 0 && n > 1){
		return 2*n - 3;
	} else if (B == 1){
		return 2*n;
	} else {
		throw new RuntimeException("Problem in Heuristic One");
	}
}

public int evaluate(State state) {
	int B = state.getB();
	int n = state.getML() + state.getCL();
	if (B == 0){
		return 2 * ((n+1)/2) - 1;
	} else if (B == 1){
		return 2 * ((n+1)/2);
	} else {
		throw new RuntimeException("Problem in Heuristic Two");
	}
```

```
}
```

The following lines presents part of the output of the program. It shows the nodes as they are being visited. Note that some nodes are visited but not expanded, which means that it was the smallest node in the fringe but it had a state that had already been expanded. Comparing the first lines of the output to the list of nodes in 2c) shows that the implementation matches the first 10 nodes visited by the A* algorithm.

```
Visiting node: [state: (3, 3, 0, 0, 0), action: null, cost: 0, depth: 0]
Expanding node: [state: (3, 3, 0, 0, 0), action: null, cost: 0, depth: 0]
Generating node: [state: (3, 2, 0, 1, 1), action: (0, 1), cost: 1, depth: 1]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Visiting node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Expanding node: [state: (2, 2, 1, 1, 1), action: (1, 1), cost: 1, depth: 1]
Generating node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Generating node: [state: (3, 3, 0, 0, 0), action: (1, 1), cost: 2, depth: 2]
Visiting node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Expanding node: [state: (3, 1, 0, 2, 1), action: (0, 2), cost: 1, depth: 1]
Generating node: [state: (3, 2, 0, 1, 0), action: (0, 1), cost: 2, depth: 2]
Generating node: [state: (3, 3, 0, 0, 0), action: (0, 2), cost: 2, depth: 2]
Visiting node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Expanding node: [state: (3, 2, 0, 1, 0), action: (1, 0), cost: 2, depth: 2]
Generating node: [state: (2, 2, 1, 1, 1), action: (1, 0), cost: 3, depth: 3]
Generating node: [state: (3, 1, 0, 2, 1), action: (0, 1), cost: 3, depth: 3]
Generating node: [state: (3, 0, 0, 3, 1), action: (0, 2), cost: 3, depth: 3]
Visiting node: [state: (3, 2, 0, 1, 0), action: (0, 1), cost: 2, depth: 2]
Repeated state: [state: (3, 2, 0, 1, 0), action: (0, 1), cost: 2, depth: 2]
Visiting node: [state: (3, 0, 0, 3, 1), action: (0, 2), cost: 3, depth: 3]
Expanding node: [state: (3, 0, 0, 3, 1), action: (0, 2), cost: 3, depth: 3]
Generating node: [state: (3, 1, 0, 2, 0), action: (0, 1), cost: 4, depth: 4]
Generating node: [state: (3, 2, 0, 1, 0), action: (0, 2), cost: 4, depth: 4]
Visiting node: [state: (3, 1, 0, 2, 0), action: (0, 1), cost: 4, depth: 4]
Expanding node: [state: (3, 1, 0, 2, 0), action: (0, 1), cost: 4, depth: 4]
Generating node: [state: (1, 1, 2, 2, 1), action: (2, 0), cost: 5, depth: 5]
Generating node: [state: (3, 0, 0, 3, 1), action: (0, 1), cost: 5, depth: 5]
Visiting node: [state: (1, 1, 2, 2, 1), action: (2, 0), cost: 5, depth: 5]
Expanding node: [state: (1, 1, 2, 2, 1), action: (2, 0), cost: 5, depth: 5]
Generating node: [state: (3, 1, 0, 2, 0), action: (2, 0), cost: 6, depth: 6]
Generating node: [state: (2, 2, 1, 1, 0), action: (1, 1), cost: 6, depth: 6]
Visiting node: [state: (3, 0, 0, 3, 1), action: (0, 1), cost: 5, depth: 5]
Repeated state: [state: (3, 0, 0, 3, 1), action: (0, 1), cost: 5, depth: 5]
Visiting node: [state: (2, 2, 1, 1, 0), action: (1, 1), cost: 6, depth: 6]
Expanding node: [state: (2, 2, 1, 1, 0), action: (1, 1), cost: 6, depth: 6]
Generating node: [state: (0, 2, 3, 1, 1), action: (2, 0), cost: 7, depth: 7]
Generating node: [state: (1, 1, 2, 2, 1), action: (1, 1), cost: 7, depth: 7]
.
.
.
Visiting node: [state: (0, 2, 3, 1, 0), action: (0, 1), cost: 10, depth: 10]
Expanding node: [state: (0, 2, 3, 1, 0), action: (0, 1), cost: 10, depth: 10]
Generating node: [state: (0, 1, 3, 2, 1), action: (0, 1), cost: 11, depth: 11]
Generating node: [state: (0, 0, 3, 3, 1), action: (0, 2), cost: 11, depth: 11]
Visiting node: [state: (1, 1, 2, 2, 0), action: (1, 0), cost: 10, depth: 10]
Expanding node: [state: (1, 1, 2, 2, 0), action: (1, 0), cost: 10, depth: 10]
Generating node: [state: (0, 1, 3, 2, 1), action: (1, 0), cost: 11, depth: 11]
```

```
Generating node: [state: (0, 0, 3, 3, 1), action: (1, 1), cost: 11, depth: 11]
Move 0 missionaries and 2 cannibals to the right side
Move 0 missionaries and 1 cannibals to the left side
Move 0 missionaries and 2 cannibals to the right side
Move 0 missionaries and 1 cannibals to the left side
Move 2 missionaries and 0 cannibals to the right side
Move 1 missionaries and 1 cannibals to the left side
Move 2 missionaries and 0 cannibals to the right side
Move 0 missionaries and 1 cannibals to the left side
Move 0 missionaries and 2 cannibals to the right side
Move 1 missionaries and 0 cannibals to the left side
Move 1 missionaries and 1 cannibals to the right side
Total number of moves: 11
Depth of the solution: 11
Maximum length of the fringe: 9
Number of nodes generated: 30
Number of nodes expanded: 14
```

Results

The following tables present the results obtained in the experiments. Note that the number of moves is not shown, because it is always optimal and corresponds to the depth the solution was found. No solution was found for the <4 4 2> scenario.

Scenario	IDS		
	Depth	Generated	Expanded
<3 3 2>	11	20751	9010
<4 4 2>	Algorithm does not stop		
<4 4 3>	9	24849	8132
<6 5 2>	19	326699294	119121968
<6 6 4>	9	216299	53879
<7 7 4>	11	3976015	979271

Scenario	A* h1				A* h2			
	Depth	Max Fringe	Generated	Expanded	Depth	Max Fringe	Generated	Expanded
<3 3 2>	11	9	30	14	11	5	30	14
<4 4 2>	N.A.	11	22	11	N.A.	7	22	11
<4 4 3>	9	22	32	11	9	22	42	15
<6 5 2>	19	41	75	31	19	13	77	33
<6 6 4>	9	37	50	14	9	46	65	17
<7 7 4>	11	42	64	18	11	52	77	22

The space requirements were not directly collected in the program, because it is difficult to keep track of the number of nodes used in each step of the algorithm. However, we can infer how much space each algorithm uses. Iterative deepening search does not need to keep all the nodes in a level, but only those that are children of nodes that are part of the path from the current node to the root. For the <3, 3, 2> scenario, the branching factor is 3 and the depth is 11, which means that the algorithm must keep at most 33 nodes in memory.

On the other hand, the A* algorithm keeps every node in memory, which means that it space usage is equal to the number of nodes generated.

As mentioned in 2b), the first heuristic dominates the second one. This means that A* using the first heuristic should always find the optimal solution is less steps than A* with the second heuristic. This is in fact what happens: the number of nodes expanded in every scenario is less or at least equal when using the first heuristic than when using the second one.

Discussion

The branching factor of the problem is the number of moves that can be made from each valid state, which in turn depends on the capacity of the boat. For the <3, 3, 2> scenario, this is at most 3 and 2.125 on average, as can be seen from the state space diagram.

The strenght of the iterative deepening search is that it does not use as much memory as A*. For these scenarios, however, this is not a problem because A* is not using too much space either. The major weakness of IDS is that it generating a lot of nodes without using any knowledge of the problem. It is also repeating a lot of nodes because the implementatios is not using a closed list to keep track of the visited nodes.

The strength of the A* search is that it does not visit or expand every node, but only those that help find the solution faster. The main weakness is that it requires knowledge about the problem (the heuristic), which can be tricky to generate.